УДК 004.051

Милова Е. А., Свешникова С. Ю., Ганкевич И. Г.

Ускорение обучения глубокой нейронной сети путем оптимизации алгоритма для запуска на MIC архитектуре

Рекомендовано к публикации профессором Дегтяревым А. Б.

1. Введение. Глубокой нейронной сетью называют перцептрон с более, чем одним скрытым (обучающим) слоем. Для обучения такой сети обычно применяется метод обратного распространения ошибки [1]. Метод обратного распространения ошибки — итеративный градиентный алгоритм, целью которого является минимизация ошибки при обучении нейронной сети. Итерация алгоритма состоит из трех основных шагов-функций: dnnForward прогоняет через сеть обучающую выборку, получая на выходе некоторый результат; dnnBackward вычисляет ошибку, затем в каждом слое сети, начиная с предпоследнего, для каждого узла вычисляет коррекцию весовых коэффициентов; dnnUpdate обновляет веса нейронов в соответствии с вычисленной ранее поправкой. Обучение сети заканчивается, когда ошибка достигает заданного минимально допустимого значения. Такая сеть показывает прекрасные результаты во многих областях, в том числе таких, как распознавание изображений и голоса. Однако ее недостатком является очень длительный процесс обучения. В связи с этим, решено исследовать вопрос об эффективности работы такого вида сетей на параллельных вычислительных архитектурах. Для тестирования взята нейронная сеть из 8 слоев (1 входящий, 6 скрытых, 1 выходящий). Для анализа результатов выбраны такие параметры, как скорость обучения нейронной сети и точность распознавания объектов.

Запуск задачи производился на процессоре Intel Xeon (спецификация указана в таблице 1). Сначала был произведен запуск задачи

Милова Евгения Андреевна – студент, Санкт-Петербургский государственный университет; e-mail: milova.evg@gmail.com, тел.: +7(911)038-38-35

Свешникова Светлана Юрьевна – студент, Санкт-Петербургский государственный университет; e-mail: svetasvesh@yandex.ru, тел.: +7(812)428-47-83

Ганкевич Иван Геннадьевич – аспирант, Санкт-Петербургский государственный университет; e-mail: i.gankevich@spbu.ru, тел.: +7(812)428-47-83

с использованием только одного ядра. Затем произведена оптимизация кода для запуска на параллельной архитектуре. Также принято решение протестировать эффективность MIC-архитектуры для решения данной задачи. MIC (Many Integrated Core) — архитектура, основой которой является использование большого количества вычислительных ядер архитектуры x86 в одном сопроцессоре, подключаемом к основному процессору. Характеристики используемого сопроцессора Intel Xeon Phi также указаны в таблице 1.

Таблица 1. Характеристики вычислителей, используемых для запуска задачи

Процессор	2×Intel Xeon CPU E5-2695 v2 (12 ядер, 2 потока на ядро, 2,40ГГц)
Сопроцессор	Intel Xeon Phi-5110P (60 ядер, 4 потока на ядро, 1,053ГГц)

2. Оптимизация для запуска на параллельных архитектурах. Каждое ядро процессора Intel Xeon и сопроцессора Intel Xeon Phi содержит блок векторных вычислений. За один такт процессора возможна обработка 16-ти 32-битных чисел или 8-ми 64-битных чисел. Векторизация кода при обработке массивов дает большой потенциал для ускорения работы программы при запуске на параллельных архитектурах. Для векторизации использовалась технология Array Notation расширения Intel Cilk Plus. Intel Cilk Plus — расширение языков C и C++ для поддержки параллелизма, реализованное в компиляторе Intel.

Для работы с массивом вместо цикла for в Array Notation используется конструкция `array[start_index : length]`. Например, следующий код к каждому i-му элементу массива W прибавляет i-й элемент массива Wdelta

```
W[0:count] += Wdelta[0:count];
```

С помощью Array Notation можно векторизовать выполнение и более сложных операций. Поиск максимального элемента в массиве выполняется при помощи выражения `__sec_reduce_max`

```
const float max = __sec_reduce_max(in_vec[base:ncols]);
```

Суммирование элементов массива — при помощи `__sec_reduce_add`

```
const float sumexp = __sec_reduce_add(in_vec[base:ncols]);
```

После проведенной векторизации код был запущен на процессоре Intel Xeon на 12 ядрах (24 потока). Программа ускорилась в 14,5 раз, по сравнению с запуском невекторизованного кода на 1 ядре.

3. Портирование кода на архитектуру MIC. Для работы с Intel Xeon Phi была использована offload-модель передачи данных. В режиме offload блок кода, выделенный директивой `#pragma offload target (mic)`, выполняется на сопроцессоре, остальной код выполняется на основном процессоре. Также для каждой переменной необходимо указать сопроцессору размер выделяемой под нее памяти. Режим offload поддерживает две модели передачи данных: явную и неявную.

3.1. Явная модель передачи данных. При использовании явной модели программист указывает, какие именно переменные должны быть скопированы на сопроцессор. Также указывается направление копирования. Достоинством данной модели является возможность успешной компиляции кода любым компилятором, а не только Intel Compiler. Неизвестные директивы будут просто проигнорированы, без генерации ошибок, код будет скомпилирован для работы только на центральном процессоре.

Таблица 2. Сравнение времени работы и точности обучения

Арх.	Версия программы	Кол-во потоков	Время, сек.	Ускорение	Точность
x86	Исходная	1	7952	× 1	19,19
x86	Параллельная	48	542	×14,7	18,99
MIC	Offload	240	6889	× 1,2	20,05
MIC	Cilk	240	589	×13,5	20,05

Функции обучения нейронной сети вызываются внутри двух вложенных циклов. Внутренний цикл был помечен для выполнения на сопроцессоре. К сожалению, у явной модели работы с памятью есть существенный недостаток. Она поддерживает лишь побитовое копирование данных. В данной программе все характеристики нейронной сети содержатся в структуре nodeArg, содержащей поля-указатели. Она указывается в качестве аргумента для отправляемых на исполнение сопроцессору функций. Для корректного копирования структуры nodeArg на сопроцессор необходимо скопировать по отдельности каждое ее поле, и уже на сопроцессоре собрать структуру заново.

Запуск на кластере показал, что такая модель передачи данных не подходит для данной задачи. Программа выполняется лишь не-

много быстрее, чем на одном ядре процессора и в 12 раз медленнее, чем на всех ядрах (таблица 2). В связи с этим, было принято решение использовать неявную модель передачи данных на сопроцессор.

3.2. Неявная модель передачи данных. Основным принципом работы неявной модели является использование разделяемой между CPU и MIC памяти в рамках единого виртуального адресного пространства. Данный подход позволяет работать со сложными типами данных. Таким образом отпадает ограничение, связанное с побитовым копированием, возникающее при использование явной модели. Преобразование программы осуществлялось следующим образом.

1. Используемые данные отмечены ключевым словом `_Cilk_shared`, которое позволяет размещать их в разделяемой памяти.

   ```
   nodeArg.d_B[i-1]=(_Cilk_shared
   float*)_Offload_shared_malloc(N*sizeof(float));
   ```

2. Используемые внутри цикла обучения функции также отмечены как разделяемые:

   ```
   #pragma offload_attribute (push, _Cilk_shared)
   ...
   #pragma offload_attribute (pop)
   ```

3. Создана отдельная функция для цикла обучения нейронной сети для ее использования в разделяемой памяти:

   ```
   _Cilk_shared void dnn(NodeArg&, ChunkContainer&) {...}
   ```

4. Функция, отправляемая для выполнения на сопроцессор помечена командой `_Cilk_offload`:

   ```
   _Cilk_offload dnn(nodeArg, oneChunk);
   ```

Стоит отметить, что неявная схема работы оказалась более проста в использовании, по сравнению с явной схемой, и позволила достичь приемлемого времени обучения. Получено ускорение времени

работы программы в 13,5 раз по сравнению с последовательной версией.

4. Выводы. Проведено тестирование задачи обучения глубоких нейронных сетей на различных вычислительных архитектурах. Результаты представлены в таблице 2. Версия для MIC не дает прироста производительности по сравнению с параллельной версией для процессора. На это влияет много факторов, которые связаны как с особенностями алгоритма, так и с теми ограничениями, которые были поставлены при решении данной задачи. Итеративность выполнения алгоритма не дает большого потенциала для распараллеливания. Оптимизации поддается только каждый его шаг, связанный с вычислениями на матрице. Полученное ускорение в 13,5 раз при запуске на MIC архитектуре по сравнению с последовательной версией в целом кореллирует с результатами, полученными в других исследованиях [2, 3]. Кроме того, не был рассмотрен native-режим работы с сопроцессором, при котором весь код запускается на сопроцессоре без использования основного процессора. Возможно, это позволит добиться большего ускорения, но данный вопрос оставлен для последующих исследований.

5. Заключение. Исследован вопрос о возможности ускорения работы нейронных сетей с последовательным алгоритмом обучения, произведена оптимизация алгоритма для параллельных архитектур, указаны причины, влияющие на эффективность распараллеливания данной задачи. Также рассмотрен вопрос эффективности MIC-архитектуры для решения данной задачи.

Литература

1. Bengio Y. Learning deep architectures for AI // Foundations and trends in Machine Learning. 2009. Vol. 2, No 1. P. 1–127.

2. Viebke A. Accelerated Deep Learning using Intel Xeon Phi: Ph.D. dissertation, Linnaeus University, 2015.

3. Dixon M., Klabjan D., Bang J. H. Implementing deep neural networks for financial market prediction on the Intel Xeon Phi // Proceedings of the 8^{th} Workshop on High Performance Computational Finance. 2015. Article No 6.